Centre Informatique pour les Lettres et les Sciences Humaines

C++ : Leçon 14
Privilèges d'accès, `this` et amitié

1 - Notion d'interface 2

2 - Les privilèges d'accès `public:` et `protected:` 3

- Accès aux membres `public:` 3
- Accès aux membre `protected:` 4
- Comment déterminer quels membres doivent rester publics 5

3 - Comment une fonction membre accède aux membres de l'instance au titre de laquelle elle est invoquée : le pointeur `this` 6

4 - Amitié 8

- Fonctions amies d'une classe 8
- Classes amies 9
- Du bon usage de l'amitié 9

5 - Bon, c'est gentil tout ça, mais ça fait quand même 8 pages. Qu'est-ce que je dois vraiment en retenir ? 9

Un des avantages déterminants de l'utilisation des classes est lié au regroupement que celles-ci opèrent entre les données et les fonctions qui les manipulent. Cette notion nous est familière depuis la Leçon 2, mais nous ne nous sommes pas encore réellement attardés sur les moyens que C++ nous propose pour systématiser cette approche. Bien que ces moyens soient simples et faciles à utiliser, ils ne trouvent leur pleine efficacité que si l'usage qui en est fait est en harmonie avec le style de programmation en vue duquel ils ont été conçus. L'objet de cette Leçon est donc de présenter à la fois la façon dont ces outils *peuvent* être utilisés (c'est à dire comment écrire du code que le compilateur acceptera) et la façon dont ils *doivent* être utilisés (c'est à dire comment rendre leur usage réellement utile).

1 - Notion d'interface

Comme nous l'avons souligné dès la Leçon 1, la façon correcte d'appliquer un traitement élémentaire sur des données dépend de façon très étroite du système de codage utilisé pour représenter ces données en mémoire. La façon dont ces traitements doivent s'enchaîner pour produire le résultat souhaité, en revanche, répond à une logique différente, liée au problème que nous cherchons à résoudre. Un des postulats de base de la programmation "orientée objets" est qu'il est préférable d'isoler nettement cette description des traitements que nous souhaitons voir exécuter dans un but particulier de la description, plus générale, de la façon dont une opération élémentaire peut être effectuée sur un certain type de données.

Cette volonté d'isolement porte le doux nom de principe d'encapsulation. Avec quelques autres termes tels que "héritage" et "polymorphisme", le mot encapsulation forme le vocabulaire indispensable à toute conversation mondaine en matière de programmation "orientée objets".

Sous une forme rudimentaire, cette idée est acceptée de tous et incorporée très concrètement dans pratiquement tous les langages de programmation. Lorsque nous souhaitons décrire la procédure permettant de calculer la moyenne d'une série de valeurs, par exemple, il est clair que nous voulons pouvoir dire quelque chose comme :

```
a) faire la somme de toutes les valeurs
b) diviser le résultat ainsi obtenu par le nombre de valeurs
```

Il ne serait pas réaliste d'exiger que la description de la tâche "calcul de la moyenne" prenne en compte le détail des opérations nécessaires pour faire la somme de deux valeurs.

Lorsqu'on s'en tient aux types de données "prédéfinis", la distinction entre ces deux descriptions est faite de façon si radicale qu'on oublie de la remarquer : décrire la procédure de calcul de la moyenne relève (généralement) d'un programme qu'il faut écrire, alors que la description de la méthode d'addition relève du langage lui-même[1]. Comme les programmeurs ne voient jamais aucun texte source correspondant à la procédure d'addition, celle-ci est parfaitement et définitivement isolée de l'usage qui est fait d'elle dans les textes sources écrits par les programmeurs en question. La technique établit ici une distinction nette entre les "utilisateurs" d'un langage et les "concepteurs" de ce langage, bien qu'il s'agisse, dans l'un et l'autre cas, de programmeurs.

Lorsqu'un langage permet aux "utilisateurs" de créer de nouveaux types de données, cette distinction s'estompe : créer de nouveaux types, c'est étendre le langage, ce qui s'approche assez vite d'une activité de conception d'un nouveau langage. Toute la question est alors de savoir comment est maintenue la distinction entre "utilisation" et "conception", et la réponse qui semble actuellement remporter le plus de succès est : grâce à l'encapsulation permise par les langages orientés objets.

Parmi les autres réponses possibles à cette question, il en est une très simple (et, donc, très attrayante) qui s'est malheureusement révélée le plus souvent inadéquate. Elle consiste tout bonnement à ne rien faire du tout pour maintenir la distinction entre conception et utilisation. Un langage tel que C, par exemple, permet tout à fait de créer de nouveaux types : les classes du C++ ne sont guère que les `struct` du C auxquelles ont été ajoutées des fonctions membre. Cette adjonction n'a toutefois rien d'anodin, car les fonctions membre (et divers dispositifs rendus possibles par leur présence) permettent d'établir une distinction claire entre le code "utilisateur" d'un type de donnée (qui ne dépend que de la présence de l'information et de son exactitude) et le code "de conception" de ce type (qui dépend étroitement de la façon dont l'information est représentée en mémoire). En négligeant cette distinction, le langage C renonce

[1] Ou, plus exactement, du compilateur ou de l'interpréteur utilisé pour mettre en œuvre le langage.

à circonscrire le code dont le fonctionnement correct est remis en cause lorsqu'une modification est apportée à un type. Modifier[2] un programme écrit en C (ou dans un langage similaire) est une entreprise qui a depuis longtemps acquis une réputation détestable, notamment parce qu'il est très difficile de mesurer les conséquences exactes de chaque intervention effectuée sur le code.

> Il s'agit là d'un problème qui ne doit pas être sous-estimé, même dans le contexte d'un programme de petite taille. Qu'il s'agisse d'individus ou de sociétés multinationales, les auteurs de programmes cherchent le plus souvent à aller jusqu'aux limites de leurs compétences, et ils ont donc tout intérêt à utiliser tous les systèmes de sécurité disponibles. Un amateur débutant n'entreprendra sans doute que des projets modestes, mais il n'en viendra à bout qu'au prix de multiples remises en cause de sa vision initiale, et l'encapsulation lui est aussi utile qu'à un groupe d'experts confronté à un projet de plus grande envergure.

Grâce à leurs fonctions membre, les classes permettent d'établir une distinction très marquée entre le code qui définit une classe (les fonctions membre, principalement) et le code qui se contente d'utiliser cette classe (en l'instanciant, le plus souvent). Cette distinction peut être renforcée par la mise en place de règles claires, dont le compilateur garantira le respect, qui définissent l'usage qui peut être fait de la classe. En C++, ces règles d'usage d'une classe sont essentiellement instituées en limitant l'accès à certains des membres.

L'ensemble des membres dont l'accès reste offert aux utilisateurs de la classe constitue l'interface de celle-ci.

2 - Les privilèges d'accès `public:` et `protected:`

Dans la définition d'une classe, les spécificateurs `public:` et `protected:` permettent d'indiquer dans quelles conditions les variables et fonctions membre dont les déclarations suivent peuvent être utilisées.

> L'usage de ces spécificateurs est très libre : chacun d'entre eux peut apparaître autant de fois que nécessaire, et s'applique jusqu'à ce que le spécificateur contraire en annule l'effet. Les programmeurs préfèrent cependant le plus souvent regrouper les membres soumis aux même règles d'accès, ce qui revient à diviser la définition de la classe en une partie `public:` et une partie `protected:`. Chacun de ces spécificateur n'apparaît alors qu'une seule fois.

Accès aux membres `public:`

La compréhension de l'effet du spécificateur d'accès `public:` pose d'autant moins de problème qu'il s'agit là du privilège dont nous avons systématiquement bénéficié jusqu'à présent. Rappelons donc simplement que les membres déclarés comme étant `public:` peuvent être utilisés librement, et que cette liberté prend deux formes :

- Les fonctions membre accèdent aux membres de l'instance au titre de laquelle elles sont invoquées en utilisant simplement le nom de ce membre.
- Dans tous les autres cas, l'accès au membre suppose l'application du sélecteur "point" à une instance de la classe, ou du sélecteur "flèche" à un pointeur sur une instance.

Les différents cas possibles sont illustrés dans l'exemple suivant :

```
//définition de la classe (normalement contenu dans CExemple_1.h)
class CExemple_1
{ public:
   int m_entier;
   void fonctionMembre();
};
```

```
//définition de la fonction membre (normalement contenu dans CExemple_1.cpp)
void CExemple_1::fonctionMembre()
{
//accès au membre de l'instance au titre de laquelle la fonction est invoquée
m_entier = 12;
```

[2] Ou, simplement, corriger...

```
//accès au membre d'une autre instance
CExemple_1 uneAutreInstance;
uneAutreInstance.m_entier = 7;
CExemple_1 * ptrSurInstance = & uneAutreInstance;
ptrSurInstance->m_entier = 6;
}
```

On remarque dans cet exemple que, lorsqu'il s'agit d'accéder à un membre d'une autre instance que celle au titre de laquelle elle est invoquée, une fonction membre ne jouit d'aucun privilège particulier : il lui faut évidemment spécifier sur quelle instance elle veut opérer, et appliquer un sélecteur. Elle se trouve donc exactement dans le même cas qu'une fonction non membre de la classe qui comporterait un fragment de code tel que :

```
CExemple uneInstance;
uneInstance.fonctionMembre();
```

Accès aux membre `protected:`

A la différence des membres public:,

> L'accès aux membres protégés d'une classe est réservé aux fonctions membre de cette classe.

En d'autres termes, pour une fonction membre, le fait que les variables membre de la même classe soient publiques ou protégées ne change rien : la fonction accède aux membres de l'instance au titre de laquelle elle est invoquée en utilisant leur nom, et aux membres des autres instances en appliquant un sélecteur sur le nom de ces instances.

```
//définition de la classe (normalement contenu dans CExemple_2.h)
class CExemple_2
{
public:
   void fonctionMembre();
protected:
   int m_entier;
};
```

```
//définition de la fonction membre (normalement contenu dans CExemple_2.cpp)
void CExemple_2::fonctionMembre()
{//accès au membre de l'instance au titre de laquelle la fonction est invoquée
m_entier = 12;
//accès au membre protégé d'une autre instance
CExemple uneAutreInstance;
uneAutreInstance.m_entier = 7;
CExemple * ptrSurInstance = & uneAutreInstance;
ptrSurInstance->m_entier = 6;
}
```

L'accès aux membres protégés est normalement interdit aux fonctions extérieures à la classe :

```
void demo()
{
CExemple uneInstance ;
uneInstance.m_entier = 12 ;          //ERREUR. Compilation impossible !
uneInstance.fonctionMembre() ;       //OK : la fonction membre est public:
}
```

La seule façon de modifier le contenu d'une variable membre protégée est donc d'utiliser une fonction membre qui effectue la modification. De même, la seule façon de connaître le contenu d'une variable membre protégée est d'invoquer une fonction membre qui renvoie l'information. Ces limitations sont la clé de voûte du système qui permet à C++ de maintenir la distinction entre le code utilisant une classe et le code définissant celle-ci : si une variable membre est protégée, seules les fonctions membre de la même classe peuvent être concernées par une modification (suppression, changement de type, etc.) concernant la variable.

> La protection est fondamentalement un dispositif de limitation de la propagation des conséquences des modifications apportées au code.

Comment déterminer quels membres doivent rester publics

Etant donné que l'usage des membre publics est laissé libre, ces membres constituent l'interface de la classe. Puisque ces membres déterminent ce qu'il est possible de faire avec une instance de la classe, leur liste constitue une sorte de "mode d'emploi" de la classe pour les programmeurs qui utilisent celle-ci. La consultation de cette liste permet de se faire une idée de "ce dont la classe est capable" et, donc, de comment on peut s'y prendre pour obtenir le résultat souhaité. Une modification du fonctionnement interne de la classe (opérée par les programmeurs responsables de la conception de celle-ci) qui ne remet pas en cause le fonctionnement de l'interface (tel qu'il a été décrit aux programmeurs utilisateurs de la classe) restera sans conséquence sur le bon fonctionnement du programme.

L'interface peut être vue comme un contrat liant concepteurs et utilisateurs d'une classe. Les premiers annoncent certaines caractéristiques, et ne s'autorisent ensuite que des extensions : de nouvelles possibilités peuvent être ajoutées, mais tout ce qui était possible doit le rester. Les utilisateurs, de leur côté, s'engagent à ne rien utiliser qui ne fasse partie de l'interface de la classe. Le langage les aide à respecter ce contrat en refusant de compiler toute ligne de code qui ne s'y conformerait pas.

Une fois l'idée de ce contrat admise, comment détermine-t-on, concrètement, quels membres peuvent être protégés et quels membres doivent rester publics ? Ce problème est facile à résoudre, pour autant qu'on garde présent à l'esprit l'objectif poursuivi, qui est que l'interface doit permettre d'utiliser les fonctionnalités offertes par la classe, tout en masquant le plus possible la manière dont ces fonctionnalités sont obtenues. Quelques règles générales[3] permettent d'orienter dans la bonne direction le développement d'une classe. La plus importante de ces règles est sans doute que

> Toutes les variables membre doivent être protégées.

En effet, la façon dont l'information est codée en mémoire ne concerne que les concepteurs de la classe, et le code utilisateur ne doit pas être autorisé à en dépendre. La protection des variables membre permet aux concepteurs de modifier ces variables (leur nom, leur type, leur nombre...) sans que ces modifications affectent le code utilisateur.

Pour ce qui est des fonctions membre, la situation est bien entendu un peu plus complexe. Un certain nombre de ces fonctions doivent sans doute rester publiques, faute de quoi la classe risque de devenir inutilisable. Il est, par exemple, assez évident que l'interface d'une classe destinée à être instanciée directement doit comporter au moins un constructeur ! D'une façon générale, les fonctions membre publiques seront celles qui offrent les services qui sont la raison d'être de la classe, alors que les fonctions "de service" ont vocation à rester protégées. Une fonction créée pour éviter la duplication de code dans plusieurs fonctions membre est un exemple typique de fonction membre qui doit être exclue de l'interface : son rôle et la façon dont elle doit être utilisée ont toutes chances de s'avérer incompréhensibles pour un simple utilisateur de la classe. Il est, par ailleurs, probable que les paramètres d'une telle fonction dépendent étroitement des choix faits au niveau des variables membre, et il est donc tout à fait légitime que l'évolution de la classe remette en cause ces paramètres. Rendre une telle fonction publique ne ferait qu'obscurcir l'usage de la classe et compromettre l'encapsulation en permettant au code utilisateur de dépendre de la nature des variables membre.

Lors de la mise au point d'une classe, il faut prendre en compte le fait que le regard que les utilisateurs porteront sur la classe doit être différent du regard du concepteur. En particulier,

> Les fonctions de l'interface doivent porter des nom évoquant la *signification* de leur action, et non la *nature* de celle-ci.

Dans bien des cas, les programmeurs débutants ont tout intérêt à écrire d'abord le code utilisateur d'une classe, et seulement ensuite le code définissant la classe en question. Le simple relevé des fonctions dont ils ont eu besoin de supposer la présence lors de l'écriture du code utilisateur décrit l'interface de la classe à réaliser. Toutes les autres fonctions membre qu'il s'avère ensuite nécessaire d'écrire pour conférer à la classe les propriétés souhaitées pourront vraisemblablement être protégées, puisque le code utilisateur a pu être écrit sans les invoquer. Comme AUCUNE variable membre ne doit faire partie de l'interface, un autre

[3] Une règle générale est une règle qui ne doit être violée que lorsqu'on sait exactement pourquoi.

avantage de cette façon de procéder est qu'elle conduit à ne prendre des décisions sur les variables membre qu'après un inventaire complet des fonctionnalités qui seront nécessaires. Le choix des structures de données (tableaux, listes, etc....) et des modes de gestion (allocation dynamique ou non) utilisés a alors toutes les chances d'être bien plus judicieux que s'il avait été fait a priori.

3 - Comment une fonction membre accède aux membres de l'instance au titre de laquelle elle est invoquée : le pointeur this

Lorsqu'une fonction membre est appelée en appliquant un sélecteur à une instance (ou à un pointeur sur une instance), cette fonction accède "par défaut" aux membres de l'instance en question. En d'autres termes, toute ligne de la fonction qui utilise le nom d'un membre de la classe sans l'associer à une instance particulière accède en fait à l'instance pour laquelle la fonction a été appelée.

La classe CExemple_1, déjà présentée plus haut, illustre clairement ce phénomène : la fonctionMembre accède tout d'abord à la variable membre de l'instance au titre de laquelle elle est appelée, et ensuite à la variable membre d'uneAutreInstance.

```
//définition de la classe (normalement contenu dans CExemple_1.h)
class CExemple_1
{public:
   int m_entier;
   void fonctionMembre();
};
//définition de la fonction membre (normalement contenu dans CExemple_1.cpp)
void CExemple_1::fonctionMembre()
{
//accès au membre de l'instance au titre de laquelle la fonction est invoquée
m_entier = 12;
//accès au membre d'une autre instance
CExemple_1 uneAutreInstance;
uneAutreInstance.m_entier = 7;
CExemple_1 * ptrSurInstance = & uneAutreInstance;
ptrSurInstance->m_entier = 6;
}
```

Si l'on y réfléchit, le premier de ces accès présente une particularité assez curieuse. En effet, bien que la variable en cause soit désignée par son nom (m_entier), il est très probable que des exécutions successives de la fonction se traduisent par des accès à différentes variables. La même ligne de code opère effectivement sur des variables différentes lorsque la fonction est appelée au titre d'instances différentes :

```
CExemple_1 a;
CExemble_1 b;
a.fonctionMembre(); //a.m_entier reçoit la valeur 12
b.fonctionMembre(); //b.m_entier reçoit la valeur 12
```

Cet aspect essentiel du fonctionnement des classes peut être conceptualisé de deux façons différentes. La première façon consiste à imaginer que chaque instance possède son propre exemplaire de la fonction membre, ce qui revient à dire que les lignes

```
a.fonctionMembre();
```

et

```
b.fonctionMembre();
```

n'appellent pas la même fonction. Il devient alors assez facile de comprendre que la ligne

```
m_entier = 12;
```

(qui figure dans les deux fonctions) n'opère pas sur la même variable dans les deux cas. Vu sous cet angle, ce phénomène rappelle la situation banale de deux fonctions qui auraient chacune une variable locale que le hasard auraient rendues homonymes : personne ne s'étonnerait alors du fait que le même nom de variable permet aux deux fonctions d'accéder à des variables différentes.

Cette explication présente l'avantage d'être tout à fait conforme à l'esprit dans lequel le système des classes et des instances a été conçu et doit être utilisé. Elle présente toutefois un léger inconvénient : elle n'est pas du tout conforme à la réalité.

En effet, l'idée selon laquelle chaque instance pourrait posséder sa propre copie de chaque fonction membre paraît totalement irréaliste dès que l'on se préoccupe de l'efficacité du système, car la quantité de mémoire requise deviendrait rapidement prohibitive. Le langage C++ a donc recours à un artifice qui lui permet de fonctionner "comme si" chaque instance disposait de sa propre copie de chaque fonction membre[4], sans avoir pour autant à dupliquer les fonctions en questions. Cet artifice repose sur l'usage d'un paramètre caché, utilisé automatiquement lors de l'appel d'une fonction membre à l'aide d'un sélecteur. Ce paramètre est de type "pointeur sur une instance" et permet de transmettre à la fonction membre l'adresse de l'instance au titre de laquelle l'appel est effectué. La fonction membre peut ensuite utiliser ce pointeur pour "compléter" (de façon invisible) les accès aux variables membre. Si ce paramètre était rendu visible, la classe `CExemple_1` ressemblerait à ceci :

```
   //définition de la classe (normalement contenu dans CExemple_1.h)
1  class CExemple_1
2  {
3  public:
4     int m_entier;
5     void fonctionMembre(CExemple * this);
6  };
   //définition de la fonction membre (normalement contenu dans CExemple_1.cpp)
7  void CExemple_1::fonctionMembre(CExemple * this)
8  {
   //accès au membre de l'instance au titre de laquelle la fonction est invoquée
9  this->m_entier = 12;
   //accès au membre d'une autre instance
10 CExemple_1 uneAutreInstance;
11 uneAutreInstance.m_entier = 7;
12 CExemple_1 * ptrSurInstance = & uneAutreInstance;
13 ptrSurInstance->m_entier = 6;
14 }
```

et on peut imaginer que l'appel de la `fonctionMembre` se présenterait ainsi :

```
1  CExemple_1 a;
2  CExemple_1.fonctionMembre(&a);
```

Bien entendu, la transmission de ce paramètre doit rester invisible et le code présenté ci-dessus ne serait accepté par aucun compilateur.

> Le mot `this` est un mot réservé du langage C++, et ne saurait donc être utilisé pour baptiser un paramètre ordinaire. Par ailleurs, l'appel d'une fonction membre *au titre de la classe elle-même* (et non au titre d'une instance) suggéré ici exigerait qu'il s'agisse d'une fonction statique, notion que nous n'aborderons qu'à la Leçon 17.

L'existence du pointeur `this` est cependant bien réelle et, si sa transmission reste automatique et invisible, il est possible (et même parfois indispensable) de l'utiliser explicitement. Un cas classique est celui d'une fonction membre qui retourne une référence à l'objet au titre duquel elle est invoquée. Ajoutons, par exemple une fonction `ajoute()` à notre classe `CExemple_1` :

```
1  CExemple_1 & CExemple1::ajoute(int val)
2  {
3  m_entier += val ;
4  return *this ;    //this est un pointeur. Il faut donc le déréférencer pour
5  }                 //renvoyer une référence
```

Etant donné que la fonction `ajoute()` renvoie une référence à un `CExemple_1`, une expression qui l'appelle désigne un `CExemple_1`, qui peut lui-même être utilisé pour appeler une autre (ou la même) fonction membre de cette classe. Il devient alors possible d'écrire des choses comme :

[4] Ce "comme si" a cependant ses limites. Le comportement des variables locales statiques d'une fonction membre, en particulier, révèle clairement qu'il n'existe qu'un exemplaire de la fonction.

```
CExemple_1 c ;
c.m_entier = 0 ;
c.ajoute(3).ajoute(5); // c.m_entier contient maintenant 8
```

Ce type de notation s'avère très agréable à utiliser lorsque la classe concernée supporte l'arithmétique usuelle. Elle devient encore plus naturelle lorsque les fonctions membre sont désignées par des symboles (+, -, *, etc.) et utilisent une syntaxe opérateur/opérandes au lieu d'être désignées par des noms et d'utiliser la syntaxe des appels de fonction (cf. Leçon 15).

4 - Amitié

Le code qui définit le comportement associé à une classe est normalement contenu dans des fonctions membre de la classe en question. Du fait qu'elles sont membres de la classe, ces fonctions accèdent librement aux variables membre protégées, ce qui leur permet d'effectuer les traitements nécessaires. Différentes raisons peuvent toutefois nous contraindre à rejeter hors d'une classe une fonction qui contribue pourtant à la définition de celle-ci.

> Parmi les causes possibles d'un tel rejet, on peut citer des contraintes syntaxiques (certains opérateurs ne peuvent pas être pris en charge par des fonctions membre, comme nous le verrons dans la prochaine Leçon) et l'utilisation d'une fonction ayant comme paramètre un pointeur sur une fonction (cf. Leçon 21).

Lorsqu'une telle situation se présente, une classe peut accorder à une fonction qui lui est étrangère des privilèges d'accès équivalents à ceux d'une fonction membre. On dit alors que cette fonction est "amie" (friend) de la classe. Ce lien "amical" peut également être établi entre deux classes : si deux classes sont conçues conjointement, il arrive que toutes les fonctions membre de l'une aient besoin d'accéder aux variables membre de l'autre. Plutôt que d'avoir à accorder individuellement ce privilège à chacune des fonctions, il est alors possible de déclarer globalement l'une des classes comme étant "amie" de l'autre.

> Remarquez au passage que la définition de l'amitié esquissée ci-dessus implique qu'il ne s'agit pas d'une relation symétrique : lorsqu'une classe en déclare une seconde "amie", elle lui accorde des privilèges d'accès à ses membre sans obtenir aucune réciprocité. C'est seulement si la seconde classe déclare également la première comme étant "amie" que les fonctions membre de la première classe auront accès aux membres protégés de la seconde.

Fonctions amies d'une classe

L'attribution du privilège d'accès aux membres protégés d'une classe à une fonction extérieure à la classe est réalisée simplement en ajoutant à la définition de la classe une déclaration de cette fonction, précédée du mot `friend`[5]. La classe `CExemple_2` est ainsi dotée d'une fonction amie, qui est la seule à pouvoir accéder à ses membres :

```
class CExemple_2
{
protected:
   int m_valeur;
   void fonctionMembre();
friend void monAmie();
};
```

Du point de vue de la fonction `monAmie()`, tout se passe ensuite comme si tous les membres de `CExemple_2` étaient `public:`

```
void monAmie()
{
CExemple_2 uneInstance;
uneInstance.m_valeur = 5;        //bien que cette variable soit protected: !
uneInstance.fonctionMembre();    //bien que cette fonction soit protected: !
}
```

Déclarer amie une fonction qui est elle-même membre d'une autre classe ne présente aucune difficulté particulière : il suffit de veiller à utiliser le nom complet de cette fonction lors de la déclaration d'amitié :

[5] Que cette déclaration d'amitié figure dans une section `public:` ou `protected:` de la classe ne fait aucune différence

```
class CExemple_3
{
protected:
   int m_valeur;

friend void CUneAutreClasse::monAmieMembreDUneAutreClasse();
};
```

Classes amies

Pour déclarer une classe entière comme étant "amie" de la classe en cours de définition, il suffit d'insérer une déclaration de la classe amie précédée du mot "friend" :

```
class CExemple_4
{
protected:
   void fonctionMembre();
};

class CExemple_5
{
protected:
   int m_valeur;
friend class CExemple_4;
};
```

Cette relation d'amitié étant établie, une fonction membre de CExemple_4 accède librement aux membres d'une instance de CExemple_5, comme s'ils étaient tous public:

```
void CExemple_4::fonctionMembre()
{
CExemple_5 uneInstance;
uneInstance.m_valeur = 6;     //bien que cette variable soit protected: !
}
```

Du bon usage de l'amitié

L'établissement de relations d'amitié est un moyen de contourner la protection mise en place par les privilèges d'accès. Abuser de ce genre de relations peut conduire à une situation complexe présentant exactement les même propriétés que la situation simple qui aurait été obtenue en laissant tous les membres publics... Le bon sens recommande donc de n'utiliser l'amitié que lorsque des raisons sérieuses obligent à déporter hors de la classe des fonctions qui participent réellement à l'implémentation des fonctionnalités que doit offrir la classe.

Une fonction (ou une classe) qui ne fait qu'utiliser une classe ne doit jamais être autorisée à entretenir des liens d'amitié avec celle-ci.

5 - Bon, c'est gentil tout ça, mais ça fait quand même 8 pages. Qu'est-ce que je dois vraiment en retenir ?

1) Les membres d'une classe peuvent être publics ou protégés.
2) Les membres publics sont accessibles à quiconque à accès à l'instance dont ils font partie.
3) Les membres protégés ne sont accessibles qu'aux fonctions membre de la même classe.
4) L'ensemble des membre publics d'une classe constitue l'interface de cette classe.
5) L'interface d'une classe ne doit comporter aucune variable, mais seulement des fonctions.
6) Lorsqu'une fonction membre opère sur les autres membres de l'instance au titre de laquelle elle a été invoquée, elle accède à ceux-ci en déréférençant implicitement le pointeur this, un paramètre caché qui contient l'adresse de cette instance.
7) Les classes peuvent avoir comme amies des fonctions ou d'autres classes.
8) Rien n'est protégé contre une amie.